12-17-2010

Software Renovation: An In-house Perspective with Case Studies

Brian Horton
University of New Orleans

Software Renovation:

An In-house Perspective with Case Studies

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

by

Brian Horton

B.S. University of New Orleans, 2007

December 2010

Table of Contents

List of Figures

Abstract

Programs are not only a tool for the simplification or automation of everyday tasks; they also represent a significant time and money investment. A program's life may span years, or even decades, which creates certain risks for the stakeholders involved. To mitigate the risks associated with these legacy systems, software renovation can be undertaken. Software renovation can be described as a series of processes and/or tools used to modernize a legacy system, thereby preserving and maintaining the investment it represents while decreasing the risks associated with it. In this thesis, the focus is on renovation of software created by in-house development. A series of case studies will be examined to demonstrate basic renovation strategies, including the formation of goals for a renovation and exploring trade-offs between robustness, performance, and usability.

Methodologies, Code Design, Programming Paradigms, Maintenance

1. Introduction

Software is "the collection of computer programs and related data that provide the instructions telling a computer what to do."[1] Alternatively, software can be described as a series of algorithms designed to complete a task. However, both of these definitions fail to capture an important aspect of the business realities behind software. Software is also an investment of time and money, often by many individuals over the course of many years. Thus, in addition to being a useful tool for solving a problem, software also represents a business asset. Further, like most assets, software requires upkeep. As failure to maintain real estate causes the investment to devalue, ignoring proper software maintenance will, over time, not only cause the software to suffer from reduced effectiveness as an operational tool, but will also incur business risk.

Combining these definitions of software, it then follows that Software Renovation can be described as a process by which the asset value of a legacy software system is preserved and maintained. Additionally, software renovation can also be viewed as an avenue towards risk management. From a business standpoint, the objective of software renovation can then be stated as the process of preserving the investment the legacy system represents while reducing the cost to maintain the system as well as decreasing the risks associated with the system.

The Year 2000 scare (Y2K) helped to highlight the importance of software renovation. Hard data on the true cost of Y2K is scarce, but Fortune Magazine reported in February 2000 that AT&T alone "spent more than $500 million on the problem (Bing, 2000)." AT&T was not the only organization affected by Y2K, and the scare demonstrated the dangers inherent in legacy systems. The root of the Y2K scare was

[1] http://en.wikipedia.org/wiki/Computer_software

that programmers had used a 2-digit system to represent the year, and it was feared that when January 01, 2000 rolled around systems utilizing only 2-digits would be reset to January 01, 1900 instead. The 2-digit system had been widely used to save memory space. Thus, what had been a good industry practice became a software liability.

It is useful when discussing software renovation to define what sorts of software can be considered legacy systems and are likely to be candidates for software renovation. In the context of renovation, software can be broken down into three general varieties. The first variety of software is commercial software; software that is used by the parties who purchased the software packaged as products. Commercial code is not likely to be a candidate for software renovation, as it tends to be better maintained and documented due to customer demand for increased usability and bug fixes. Product replacement, often called software upgrade, is the common practice for dealing with old commercial systems. The next variety of software is research or development software, which can be described as software exploring the feasibility of a solution to a given problem. Software of the research variety is by definition new code, and is thus highly unlikely to have suffered from neglect. The final category of software is in-house software; software that is developed for use by the initial developer(s). In-house software often becomes a candidate for neglect, as it does not draw in outside funding and is thus less likely to have the budget required for proper maintenance over time. Neglect will be a primary driver for the need for software renovation, so renovation from an in-house software perspective will be the focus of this thesis.

It should be noted that there is no single software renovation process. Many technologies and methodologies can be employed in a renovation, from migrations, to reverse engineering, to code re-factories. Each legacy system has its own set of needs depending on the technologies, structures, and documentation that comprise it, as well as the set of requirements the system must meet. For example,

a piece of accounting software written in COBOL in the early 80's would have very different renovation needs from a Java-based Web interface written in the late 90's.

As each potential renovation has different needs, information discovery is a vital portion of the renovation process. Carnegie Mellon University's Software Engineering Institute has developed a broad methodology that is an excellent starting point for a renovation (Lewis 2008). This methodology has been named the Service Migration and Reuse Technique (SMART), and it describes a process for determining what legacy systems (or portions thereof) are good candidates for migration to a service. Of particular note is the Service Migration Interview Guide (SMIG) contained in SMART. It is a series of questions that helps provide a comprehensive vision of the existing system, as well as the desired renovation.

Figure 1.1: Sample SMIG questions.

Legacy System End Users
- Who are the end users of the legacy system?
- Will legacy system end users be available during the migration process?

Legacy System Owners
- Who owns the legacy system?
- If there is more than one owner, are these separate organizations?
- Will legacy system owners be available during the migration process?

Legacy System Developers and Maintainers
- Who is the developer for the legacy system?
- Are developers available to support the migration process?
- Is the maintenance group separate from the development group?
- If so, are maintainers available to support the migration process?

Organization Performing the Migration
- Are current developers or maintainers going to be performing the migration?
- If not, what organization will perform the migration?
- What is the process for bringing them up to speed on the legacy system?
- Will this organization be available during the migration planning?

Target SOA Environment Owners
- Is the target SOA environment owned and maintained by a separate organization?

If so, will representatives be available to support the migration process?

The sample questions show in **Figure 1.1** illustrate some of the basic information gathering that helps define the path a renovation is required to take. The usefulness of this pre-renovation information-gathering is useful regardless of actual renovation situation, and as such is a solid starting point for any prospective renovation.

This thesis aims to provide case studies of successful renovations, as well as to examine strategies, technologies, and pitfalls in the software renovation field. The case studies documented the problem solving processes used in dealing with the in-house legacy software of a set of university services – the Testing Services at the University of New Orleans (UNO).

2. A Survey on Software Renovation

Many methodologies and tools have been proposed, and in some cases, implemented in the field of Software Renovation. Prior to the renovation work later described in this thesis, a survey of existing technologies and methodologies was performed.

2.1 Code Factories

Of particular note amongst renovation tools are Code Factories. Code Factories are, in theory, an ideal solution when software renovation requires a language migration (the shift in use from one programming language to another). The basic idea behind a Code Factory is that source code is inputted on one end, transformations are applied by the factory, and workable code in the target language is produced. Several factors must be explored when assessing the viability of code factories. Some insight was gained in an earlier software renovation project.

First, does the code factory preserve the existing structure of the legacy system, and how much preservation is necessary or desirable? Second, does the code factory attempt to conform to best industry standards; does it produce well-formed code with human-readable variable names, among other issues? Thirdly, does the end result perform at least as well as the un-translated legacy system? Finally, does the translation reduce the risks and costs associated with the legacy system?

To address these issues, an example code factory can be employed. For the following example, a demo version of Diamond Edge's VB Converter[2] will be used to convert code from the DB Connections sample code bundle from VBCode.com.[3]

[2] http://www.tvobjects.com/

[3] http://www.vbcode.com/Asp/showzip.asp?ZipFile=http%3A%2F%2Fwww%2Evbcode%2Ecom%2Fcode%2FDBConnections05102010%2Ezip&theID=13735

Figure 2.1: Relevant section of VB source

```
Private Sub Form_Load()
    Dim SQL_StrConn_STRING As ADODB.Connection
    Set SQL_StrConn_STRING = New ADODB.Connection
    Dim MySQL_StrConn_STRING As ADODB.Connection
    Set MySQL_StrConn_STRING = New ADODB.Connection
    Dim Access_StrConn_STRING As ADODB.Connection
    Set Access_StrConn_STRING = New ADODB.Connection

    'uses SQL Native client 9.0 OLEDB Provider(1)
    'SQL Server Computer Name: "PC091581"
    'Database Name: "Rabya_ERP"
    'User: "{youruseraccount configured in SQL Server}"
    'Password: {yourpassword}
    SQL_StrConn_STRING.ConnectionString = "Provider=SQLNCLI;Data
Source=PC091581;Initial Catalog=Rabya_ERP;User Id=sa;Password=123456;"
    SQL_StrConn_STRING.Open

    'uses MySQL ODBC 5.1 Driver
    'must install MySQL ODBC 5.1 Driver
    'MySQL Server Computer Name: "{localhost means your MySQL Server runs on
your computer}"
    'Database Name: "Rabya_ERP"
    'User: "{youruseraccount configured in SQL Server}"
    'Password: {yourpassword}
    MySQL_StrConn_STRING.ConnectionString = "Driver={mySQL ODBC 5.1 Driver};
Server=localhost; Port=3306; Database=Rabya_ERP; User=sa; Password=123456;
Option=3;"
    MySQL_StrConn_STRING.Open

    'uses ACE OLEDB 12.0
```

Next, the original VB code is then converted into Java using the VB Converter.

Figure 2.2: Relevant sections of converted code

```
import diamondedge.util.*;              (1)
import diamondedge.ado.*;
import diamondedge.vb.*;
import java.awt.*;
import javax.swing.*;
import diamondedge.swing.*;
import javax.swing.event.*;
import java.awt.event.*;


.
.
.

      Connection SQL_StrConn_STRING = null;
      Connection MySQL_StrConn_STRING = null;
      Connection Access_StrConn_STRING = null;
      Recordset SQL_RSUSer = null;
```

```
        Recordset MySQL_RSUser = null;
        Recordset Access_RSUSer = null;
        SQL_StrConn_STRING = new Connection();
        MySQL_StrConn_STRING = new Connection();
        Access_StrConn_STRING = new Connection();

        //uses SQL Native client 9.0 OLEDB Provider(2)
        //SQL Server Computer Name: "PC091581"
        //Database Name: "Rabya_ERP"
        //User: "{youruseraccount configured in SQL Server}"
        //Password: {yourpassword}
        AdoUtil.registerDriver( "sun.jdbc.odbc.JdbcOdbcDriver" ); (3)
        SQL_StrConn_STRING.setConnectionString( "jdbc:odbc:pc091581", "sa",
"123456" );
        SQL_StrConn_STRING.open();

        //uses MySQL ODBC 5.1 Driver
        //must install MySQL ODBC 5.1 Driver
        //MySQL Server Computer Name: "{localhost means your MySQL Server runs
on your computer}"
        //Database Name: "Rabya_ERP"
        //User: "{youruseraccount configured in SQL Server}"
        //Password: {yourpassword}
        AdoUtil.registerDriver( "sun.jdbc.odbc.JdbcOdbcDriver" );
        MySQL_StrConn_STRING.setConnectionString( "jdbc:odbc:driver={mysql odbc
5", null, null );
        MySQL_StrConn_STRING.open();
```

The converted Java code is generally of the same structure; both the Java and the VB code start with a block of variable setup followed by examples of database access and record retrieval using three different drivers. Further, the converter retains comments and attempts to place them similarly in the structure, as demonstrated by **Figure 2.1(1)** and **Figure 2.2(2)**.

There are certain serious issues with the converted code that need to be addressed. The first issue that must be addressed is located at **Figure 2.2(1)**:

```
import diamondedge.util.*;
import diamondedge.ado.*;
import diamondedge.vb.*;
import diamondedge.swing.*;
```

The converted Java code starts with four vendor-specific packages being imported. One of the primary goals of software renovation is to reduce risks associated with a legacy system. The use of vendor-specific packages is not a reduction in risk, but is instead a risk factor. What state would the converted

system be in if the vendor was to close its doors? This risk can be mitigated by paying an additional fee to the vendor to obtain source code for the vendor-specific library.

Another issue that can be raised concerning the converted code is maintainability. To gain a grasp of possible maintenance issues, some of the vendor-specific code can be examined. At **Figure 2.2(3)**, the class AdoUtil is utilized. To be able to maintain code utilizing this class, the structure, queries, and commands contained in it must be understood. To gain this understanding, the Javadocs can be examined. The Javadocs for the vendor-specific packages are available at http://www.tvobjects.com/java/docs/, but neither diamondege.vb nor diamondedge.ado are listed. Thus, the only way to understand the classes in these packages would be to pay for and examine the source code.

Researchers working at the Tokyo Research Laboratory have shown pitfalls of other code factories in their work. Specifically, they note that Cobol-to-Java translators "produce code that that tends not to perform well in Java (Suganuma, 2008)." They further note that some of the performance issues result from a mismatch in basic structures provided by the different languages.

With these factors in consideration, code factories cannot be considered as a complete solution to language migrations. This is not to say that code factories have no place in such a renovation. Using a code factory as an intermediate solution, perhaps to produce a renovation prototype, may be feasible.

2.2 Screen Scraping

In addition to code factories, there are also Screen Scraping[4] tools available from a variety of companies. Screen Scraping tools do not change an existing legacy system, but instead set up a new front end and

[4] http://wiki.open-esb.java.net/Wiki.jsp?page=ScreenScrapingSE, http://www.jacada.com/products/winfuse/, http://www.yrrid.com/Products/LOF-Enterprise/, www.attachmate.com/Products/Host+Integration/mainframe-integration/vstream-host-integrator.htm

intermediary to form a communication link with the existing front end. The benefit of Screen Scraping tools is that they allow for quick web accessibility of legacy systems.

However, Screen Scraping tools have certain intrinsic issues. First, no modifications are made to the underlying system. This means that any existing issues with the legacy system in question go unfixed. Further, Screen Scraping tools rely on communication with a legacy system, incurring performance risks, as well as the risk of de-synchronization between the two front ends.

With these factors considered, Screen Scraping tools are better suited towards being an intermediate solution; while the legacy system undergoes real renovation, Screen Scraping tools can be employed to bridge the time gap until the migration is complete. Further, during this time, the interface created via Screen Scraping can be utilized as a prototype for the renovated interface.

2.3 Glue Method

Another possible renovation strategy is suggested and detailed in work by researchers from the National Research Institute for Mathematics and Computer Science (CWI). In their paper, *Modernizing Existing Software: A Case Study*, they detail a cut-and-paste methodology for preserving portions of a legacy system during renovation. Their approach was to, “identify and isolate components,” and then, “glue them together by writing coordinator modules (Everaars, 2004).” Using Manifold as their coordinator language, they were able to restructure sequential C code to allow for parallel and distributed application.

This approach, if not ideal in terms of resulting structural complexity, does help to preserve the investment present in the legacy system by attempting to preserve reusable sections.

2.4 Service Migrations

A variety of tools are available to assist in migrating legacy systems to a service-based structure. These tools tend to be offered by vendors with a vested interest in a single language or technology, and support the continued usage of the specific language or technology.

Micro Focus[5] offers a series of tools to assist with renovating COBOL systems. For web service migrations they offer Micro Focus Studio, which is a COBOL development studio allowing for the integration of web services, and Micro Focus Net Express, which is an application for Modernizing COBOL-based systems and includes J2EE support. Micro Focus also offers SOA Express; a tool for aggregating and exposing existing business applications as web services without changing the back end. IBM offers the WebSphere Developer for zSeries V6.0[6], which supports J2EE, CICS, IMS, COBOL, and PL/I. It is a struts-based tool for using existing business logic in MVC patterning to create a new service.

Each of these tools is narrow in focus. Their primary failing is that they are useful only for a specific renovation case; if a desired renovation path does not desire that the specific language or technology be maintained, or migration to a service-based structure is not desirable, then the tools are of no use.

2.5 Code Understanding Tools

To help facilitate a renovation project, it may be of use to employ tools to help with code understanding. One tool that may assist in code understanding is Klockwork Insight[7]; a static code parser that supports C, C++, C#, and Java. Insight is built to gather information on architecture and control flow, as well as to attempt to identify quality and security issues in code.

[5] http://www.microfocus.com
[6] http://www-01.ibm.com/software/awdtools/devzseries/support
[7] http://www.klockwork.com/products/insight.asp

Eclipse[8] may also be useful for code understanding. The extensible nature of the framework allows for modification to fit specific renovation needs. Thus, if no existing tools are desirable given the circumstances of the renovation, and the legacy system is too monolithic to work with without automated assistance, it may be of use to create Eclipse add-ons that fit the specific need.

2.6 State of Renovation Tools and Methodologies

The tools and methodologies available for software renovation are not in a good state; most suffer from serious downsides such as narrow usability, inability to address core issues, or lack of automation. As such, developing a wide-use methodology that can be applied to renovations in general has a high degree of desirability.

[8] http://www.eclipse.org

3. Subject System: UNO Testing Services

The University of New Orleans's Office of Testing Services[9] provides test-scoring services for the University, as well as for several events held at the University. The tests are read by an optical scanner that produces text files containing the information scanned from the sheets. The actual scale of the operations at Testing Services varies widely from day-to-day. During non-peak times there may be only a few instructors coming in each day to have tests graded. In contrast, during the period in which final exams are given, dozens of instructors are expected each day, with the possibility of hundreds of tests scanned and scored in the span of a week. During such events as Literary Rally and Spring Testing, the Office of Testing services is responsible for producing results for up to several hundred students in a matter of hours.

The actual scoring, reporting, and analysis of the tests were handled by a suite of programs with several common features. In general, the programs were written in C++, they were written by the same programmer, they lacked commenting, and the documentation left on the systems was haphazard and described only their operation, never their structure or problematic/bugged areas. The original programmer of these systems was no longer available as a resource to Testing Services, and the office was without a programmer at all for approximately two years. To further complicate the matter, the only student worker with significant experience with and understanding of the programs had left Testing Services as well. Thus, many of the programs used by testing services were behaving incorrectly or outright failing in their tasks.

[9] http://www.testing.uno.edu

4. Need for Renovation

Due to the lack of commenting and documentation, maintenance of the software systems at Testing Services had been insufficient and renovation of the systems utilized by the office had to be undertaken. A primary factor in the renovation path was that Testing Services was not guaranteed that there would be a programmer or other personnel qualified to perform software maintenance in the future. If qualified personnel were on hand, it was likely to be a temporary affair. The Office of Testing Services employs primarily student workers. There was a high degree of likelihood that graduate students would be employed to maintain the systems, and thus it would be expected that a given maintainer would be on hand for no more than two years. This risk ensured that code simplicity had a high degree of desirability. With these factors considered, several goals for the renovation of the software suite could be assembled:

1. **Renovated code should be simple, with the aim to reduce training needs.**

   Since there was no guarantee of regular maintenance in the future, a minimalist structure that promoted code understanding was highly desirable.

2. **Renovated code should be well-documented in both function and operation.**

   As there was no guarantee of an unbroken understanding of the systems utilized by Testing Services, ensuring thorough and complete documentation of the systems was a high priority.

3. **Renovated code should be as robust as possible.**

   Code robustness also had a high degree of desirability due to the lack of assurance that there would be anyone qualified on hand to maintain or alter the systems as new needs would arise. The program should also be able to handle users' incorrect input by displaying easy-to-understand error messages.

4. **Renovated code should be easy to operate, and should attempt to preserve the interface(s) and functionality of the current suite.**

   Testing Services already had a high degree of required training for a new employee to be fully proficient in the office (estimated at roughly five months). Thus, it was desirable to preserve interfaces and functionality where possible to reduce the need to train and retrain employees.

5. Methodologies Applied to UNO Testing Services Systems

Several methodologies were employed during the renovation work done for the Office of Testing Services. While these are by no means the only tools and methods available for use in renovation projects, they do illustrate some basic principals in the field.

5.1 Analysis

Each renovation started with an analysis. The goal of this analysis was to gather information similar to what might be gleaned via the sample questions shown in the sample SMIG questions (**Figure 1.1**). The information gathered before deciding on a renovation path consisted primarily of the following factors:

- Which parties had a stake in the system?
- Which requirements were the system not meeting?
- What level of technical expertise was available amongst the stakeholders?
- What sort of maintenance expertise would be available post-renovation?
- What language or languages were utilized in the system?
- What sort of operations did the system perform?
- How was the system structured?

This sort of analysis allowed for a unique renovation path to be defined for each legacy system. For systems where insufficient data was available for a comprehensive analysis, Black Box Testing helped to act as a gap-filler.

5.2 Renovation Strategies

With analysis of a system complete, a renovation strategy for that specific system can be devised based on the on the desired end-result of the renovation. Components in a renovation strategy can include,

but are not limited to, target language, target database, possible tool usage, re-usable components, and documentation standards. A good renovation strategy details how to migrate a legacy system from its current state to its desired state.

5.3 Expanding Prototypes

In renovations involving the replacement of code, an expanding prototype methodology was employed. Rather than attempt an entire renovation in a single pass, a prototype that encompassed only a portion of the desired functionality was developed. After testing the prototype, the next piece of desired functionality was built on top of the existing prototype. The prototype was then expanded in this fashion until it encompassed all of the desired functionality.

Boeing employed this basic methodology to successfully renovate their *Aero Grid and Paneling System* (AGPS) from a mixture of C and FORTRAN to Java (Dickens, 2002). By using a series of expanding prototypes, they were able to modernize their system, allowing for better performance and portability. Further, they were able to speed their release cycle to every four months from every one or two years.

5.4 Renovation Methodology

In summary, the renovation process can be broken down into three steps:

1. **Analysis.**
2. **Formulation of Migration Strategy**
3. **Prototyping**

The system is analyzed to determine its existing properties and the desired end properties, a migration strategy is formulated to achieve these transformations, and an expanding prototype is developed to implement the strategy.

The core elements of this strategy were developed by examining the methodologies promoted by the SMART system, the successful renovation of Boeing's AGPS software, and a survey of various proposals for renovation tools and methodologies. This methodology is directed at software renovation in general, as opposed to the SMART system's focus on migrating legacy systems to services.

6. Analysis and Conservation

Software Renovation need not necessarily indicate the replacement or modification of existing systems. A software system may fail to meet requirements, but still be able to fulfill those requirements, simply because the end users are unaware of the system's potential capabilities.

In such a case, it makes little sense to replace an existing system, as it represents an investment of both time and resources. Analysis and Conservation are then the preferred outcomes of such a renovation; the system in question should be more thoroughly understood and its useful properties preserved.

This situation is likely to be the result of poor documentation and training methods, possibly even as the result of years of miscommunication and knowledge disconnect. To begin the renovation, a candidate system must first be tested to determine its properties, and the results of these tests must be documented so that a clear understanding of the software's true capabilities may be gleaned. Once the analysis is complete, the useful properties that are (re)discovered may be again be utilized.

6.1 Conservation Case Study: TSETSE

The Office of Testing Services employs a complex program dubbed TSETSE (the meaning of the acronym is undocumented) that handles grading the various Scantron[10] forms that professors bring in to be graded. These forms come in two varieties, denoted by color, and are run through an optical scanner to produce text files containing lines of ASCII text corresponding to the bubbles a student has filled in on the form. When the forms are scanned, a key sheet is provided by the teacher and placed on the top of the forms. The key sheet is an identical form to the ones the students fill out, indicates the correct answers, and is denoted by being the first line in the text file.

[10] http://www.scantron.com/

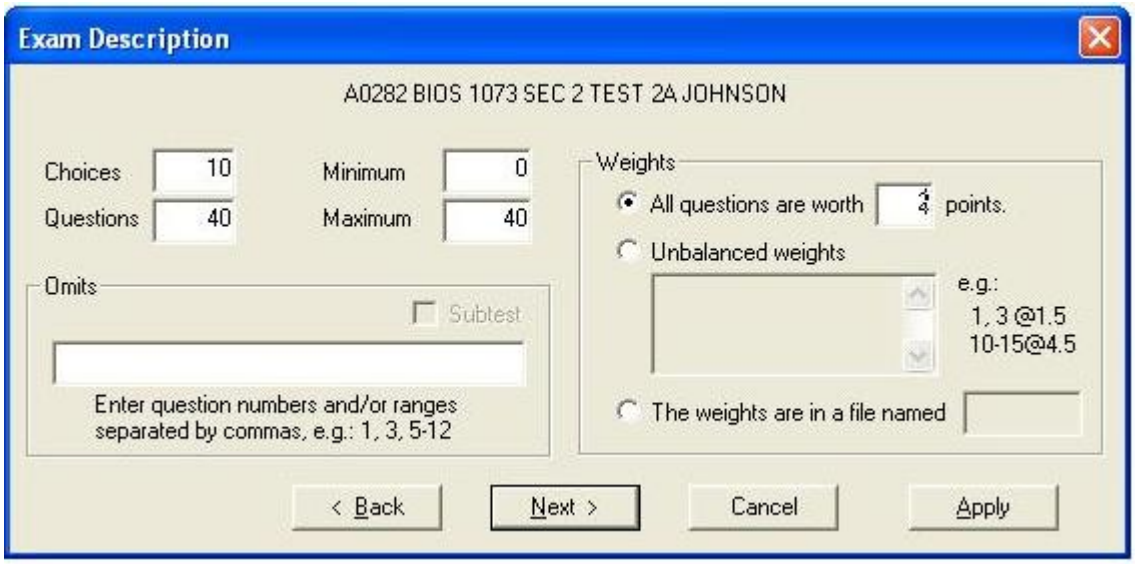
Figure 6.1: Example scanner output

```
                                 143214423212211445234532111232431341212 2
SMITH JOHN              2145568  12351134123414313 45123451234245312235132
WILSON BARBARA          3176713  5134151312532151343213  12341221341234532
JEFFERSON JEFF          4314553  143214423212211445234532111232431341212 2
```

So, in **Figure 6.1**, the first line corresponds to the answer key, and the lines below correspond to student names, numbers, and answers. The scanner produces a single string for each line, with each character representing a possible mark. If nothing has been marked, the scanner explicitly leaves a space character. For example, if a student left questions one and five blank, their answer for question thirteen would be in the same column as the key answer for question thirteen.

These text files are read by TSETSE which then offers several options for the final output.

Figure 6.2: TSETSE test option interface



As can be seen in **Figure 6.2**, the TSETSE test option interface was straightforward and offered the ability to change the minimum score on a test, what questions to omit, how much each question should be worth, etc.

While TSETSE generally performed to expectations, and was reasonably user-friendly, it was not a well-documented system. Three particular requirements proved to be a problem in meeting the business requirements of the Office of Testing Services:

1. Be able to accommodate questions worth bonus points.
2. Be able to accommodate omitted test questions.
3. Be able to accommodate unbalanced weights (some questions being worth more or less points than others).

To resolve the issues with the above requirements, a variety of information needed to be collected about TSETSE. The user's manual left by the system's original programmer proved to be sparse and unhelpful when attempting to understand how to fulfill the given requirements, and the source code itself was both reasonably complex and extremely lacking in commenting. Since these sources of information were inadequate, black box testing was employed to gather information on TSETSE.

6.1.1 Black box testing

Black-box testing is "a method of testing software that tests the functionality of an application as opposed to its internal structures or workings ."[11] The operations are assumed to be performed within a black box, and as such are not visible. To gather information about the system, tests are performed by noting what input is fed into the black box, and what corresponding output is then produced. The workings of the system are then derived based on observations of its functionality.

[11] http://en.wikipedia.org/wiki/Black-box_testing

Initial testing of the TSETSE software showed a useful feature; a test could be re-run through TSETSE with different options and the previous run would in no way bias or affect the new run. This property was tested to ensure that no assumptions were taken with without verification about the basic (and therefore easily-overlooked) traits. With this basic principle of the system established, a simple methodology for more extensive testing of TSETSE could be employed. To ascertain precisely what operations the various options were performing, the same test data was run through the system multiple times and the changes in the output were noted. Through this testing, it was ascertained that while TSETSE allowed for a great deal of precision to be entered initially, the precision was not maintained in the output, but rather rounded down to two decimal places. With these facts gathered about the system, it was then possible to set about handling the issues with the requirements previously listed.

6.1.2 Bonus points

Black Box testing TSETSE proved that rerunning a test with different parameters not biased by previous runs. Essentially, every run is fresh. Further, TSETSE allows for subtests and the omission of questions. After combining this data, a methodology for handling bonus questions could be established. The method was broken down into the following steps:

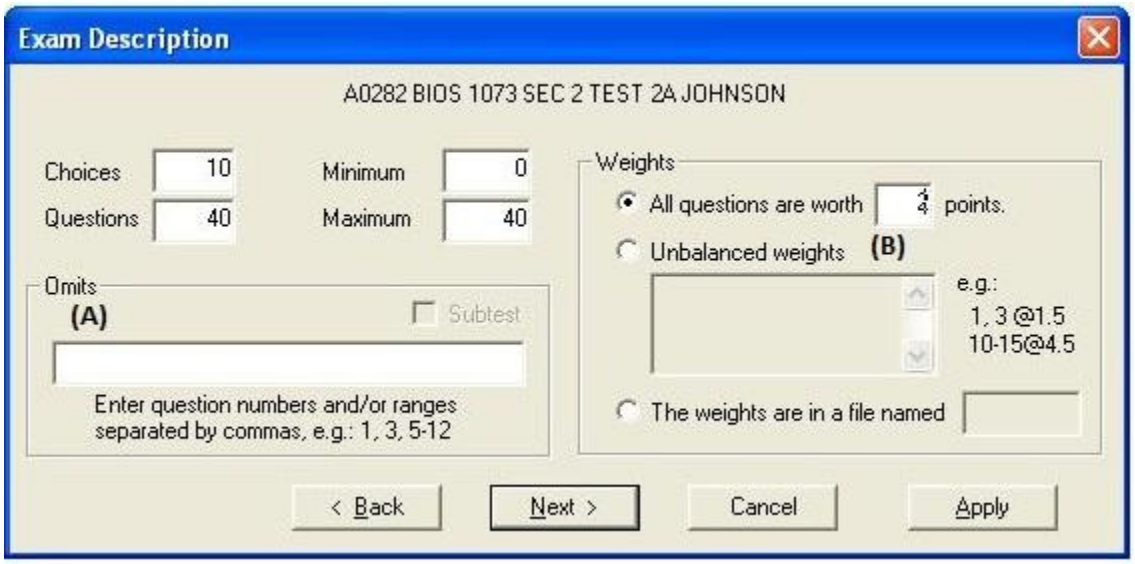
1. Omit the bonus questions and run the test as though they did not exist.
2. Re-run the test through TSETSE.
    a. Give the test a slightly different name so the previous output is not overwritten.
    b. Omit all the non-bonus questions, leaving only the bonus.
3. Present both files to the professor.

This method resulted in the professor that had requested grading receiving two files; the first containing the students' pre-bonus scores and the second containing how many bonus points each student had earned.

6.1.3 Omitted questions and unbalanced weights

Both the *unbalanced weights* and *omitted questions* requirements were separately performed by TSETSE. Further, the interface for performing these tasks was simple enough that the limited documentation was sufficient to perform each individual task.

Figure 6.3: Annotated TSETSE exam options interface



To perform unbalanced weights and omissions, different fields are employed. Field **4.3A** allows for a series of individual questions and/or sequences of questions to be listed for omission. Field **4.3B** works in a similar manner, allowing individual questions and series of questions to be assigned varying values.

However, while the system was able to perform both operations, it did not allow for unbalanced weights and the omission of questions to be performed together. To further complicate matters, testing quickly revealed that TSETSE did not allow questions to be weighted at 0.0 points, so assigning omitted questions a zero value was an insufficient solution.

Thus, the rounding properties of the system had to be exploited to allow for unbalanced weights and the omission of questions on the same test. The methodology for accomplishing this task proved to be simple once the rounding properties of the system were discovered. Since TSETSE allowed for very high initial precision, and then rounded down to two places beyond the decimal during computations, questions that were to be omitted could be given a tiny value (one one-billionth of point, for example) and would then be rounded to a value of 0.00 during computation, effectively omitting them from the test.

6.1.4 Results of TSETSE renovation

The end result of the renovation of the TSETSE software was the fulfillment of previously unmet requirements. Documentation was added, which allowed the student workers employed by Testing Services to appropriately handle instructor's requests for some questions to be worth bonus points, as well as instructor requests for exams to be graded on a metric featuring both unbalanced weights and omitted questions. Several months of use further support the ability of the TSETSE software to meet all necessary requirements.

No source code was altered or replaced during the renovation of TSETSE; the existing system was completely preserved. Black Box testing and documentation of the system's capabilities did, however, allow the system to meet previously problematic requirements. New functionality was effectively added to TSETSE, even though there were no changes made to the code. In addition, this approach greatly supported renovation goals **2** (documentation) and **4** (preservation), as the documentation of the system was greatly enhanced and the existing interface remained unchanged.

6.2 Maintenance Case Study: Teacher Evaluation System

Each semester the Office of Testing Services sends out Teacher Evaluation packets. Each non-online section of all the courses offered at the university receives a packet with forms for the students to fill out to rate the course instructor. The completed forms are returned to Testing Services for the purpose of compiling statistics on each teacher, department, and college. Further, the Teacher Evaluation system was required to be able to produce statistics on a level-wide basis (all 1000-level English courses, for example). The Teacher Evaluation system was typical of the software employed at Testing Services; it was written in C++, the documentation was near nil, and the original programmer was no longer available as a resource. In general, the system still fulfilled the requirements and was reasonably user-friendly.

The primary problem with the Teacher Evaluation system was that several departments within the university had changed acronyms, and as such were causing problems when being run through the Teacher Evaluation software. The interface for the Teacher Evaluation system offered no method for updating the acronyms, and the documentation did not provide any information on how such an update might be accomplished. Examining and documenting the source code then became the only reasonable

method for understanding the Teacher Evaluation software. Fortunately, the portion maintaining the acronyms within the Teacher Evaluation system was obvious and easily modified.

Figure 6.4: Teacher Evaluation system acronym source code

```
bool good_department (const string& d){

       bool found = false;

       string list[] =

       {  "~AS~", "AADM", "ACCT", "ANTH", "ACOR",   "BIOS", "~BA~", "ECON",
      "FIN~", "HRT~", "MANG", "MKT~", "QMBE", "CHEM", "CSCI", "DEVM", "DRCM",
      "EDAD", "EDFR", "EDGC", "EDSP", "EDHP", "EDHS", "EDUC", "EDLS", "EDCI",
      "EES~", "ENCE", "ENEE", "ENGR", "ENME", "ENMG", "NAME", "FORL", "FREN",
      "CHIN", "LAT~", "ROML", "GER~", "ITAL", "JAPN",  "MURP", "DURB",
      "URBN", "PADM", "ENGL", "FTCA", "~FA~", "GEOG", "HIST", "HUMS", "JOUR",
      "MATH", "MUS~", "PHIL", "POLI", "PSYC", "PHYS", "SOC~", "SOSC", "SPAN",
      "UNIV", "WGS~", "IDS~", "DONE"

       };


  for (int x = 0; list[x] != "DONE" && !found; x++){

  if (d == list[x]){

    found = true;

   }

  }

  return found;

}
```

With only minor changes to the source code shown in **Figure 6.4**, the Teacher Evaluation system became capable of meeting the requirements. The location of the relevant code was added to the documentation, as well as instructions on how to modify and how to add department acronyms should it become needed again.

6.2.1 Bug detection

During the course of the acronym renovation and documentation update, the table mapping departments to colleges within the university was discovered. This mapping proved to be entirely independent of the list of accepted acronyms, meaning that the mappings for the altered acronyms would be updated as well.

Figure 6.5: Sample section of the mapping function

```
string which_college (const string& department){
       typedef map <string, string> mymap;
       mymap lookup;
       lookup.insert (mymap::value_type("ACCT", "College of Business"));
       lookup.insert (mymap::value_type("~BA~", "College of Business"));
       lookup.insert (mymap::value_type("ECON", "College of Business"));
       lookup.insert (mymap::value_type("FIN~", "College of Business"));
       lookup.insert (mymap::value_type("HRT~", "College of Business"));
       lookup.insert (mymap::value_type("MANG", "College of Business"));
       lookup.insert (mymap::value_type("MKT~", "College of Business"));
       lookup.insert (mymap::value_type("QMBE", "College of Business"));
       lookup.insert (mymap::value_type("ENMG", "College of Business"));
       lookup.insert (mymap::value_type("EDFR", "College of Education"));
       lookup.insert (mymap::value_type("EDGC", "College of Education"));
       lookup.insert (mymap::value_type("EDSP", "College of Education"));
       lookup.insert (mymap::value_type("EDHP", "College of Education"));
       lookup.insert (mymap::value_type("EDAD", "College of Education"));
       lookup.insert (mymap::value_type("EDHA", "College of Education"));
      .
      .
      .
```

It was realized during the update of the function sampled in **Figure 6.5** that not only was the mapping outdated due to the updated acronyms, but the mapping was also incomplete. Roughly 40% of the departments in the university had no mapping, and thus were simply lumped into a general university average, rather than a college average. This had likely existed within the system since its inception, and thus had been going on for years. Further, there were no records of any college or department complaining about the college averages being incorrect or that some departments received a college average whereas others in the same college received general university averages.

During the course of the renovation, additional mappings were added, and the documentation was updated to make note of the mappings and explain their importance should any new departments arise. With this change, the Teacher Evaluation system again met all of the necessary requirements.

6.2.2 Results for the maintenance of the Teacher Evaluation system.

Previous to the renovation of the Teacher Evaluation system, requirements to correctly map individual departments to their corresponding colleges were being failed on a regular basis. At least part of these failures had been present, and apparently undetected, for at least several years. Post-renovation, the Teacher Evaluation system correctly mapped all departments, ensuring that every instructor and department received evaluations with correct departmental and college averages on them.

Renovation goals **2** and **4** were upheld during the renovation of the Teacher Evaluation system. Only minor updates were made to the source code, thereby preserving the existing interfaces while enhancing the functionality of the software. Useful additions were also made to the documentation, ensuring that future operators of the system would be able to prevent the same sort of failings. In addition, no complexity was added to the system in accordance with renovation goal **1**.

7. Renovation Case Study: Spring Testing/Student Orientation

The University of New Orleans periodically offers new students the opportunity to test out of generally required, low-level courses. Student Orientation acts as a subset of Spring Testing. While the same grading metrics are utilized, Spring Testing offers exams in addition to those available during Student Orientation. Spring Testing is offered only once each year, whereas Student Orientation is offered several times yearly. Spring Testing also has a higher attendance, though it still does not exceed several hundred attendees.

The Spring Testing/Student Orientation software utilized by Testing Services was programmed in C++, the code was mostly uncommented, and the documentation was exceedingly sparse. There was no one available as a resource that had worked on the source code or even operated the software. Prior to renovation, a new requirement (in the form of an additional test) had arisen for the Spring Testing/Student Orientation system. The new test was an additional English test, with a different metric utilized in the scoring process.

7.1 Forward Engineering

Forward Engineering is simply the process of creating new systems. The terminology is useful for distinguishing the process of designing and creating software from the ground up, as opposed to the methodology of Reverse Engineering, which will be discussed later in this document.

With the addition of a new requirement to the Spring Testing/Student Orientation system, software renovation became immediately necessary. The metrics for each test were hard-coded into the software, so there was no flexibility in the ability to grade tests. Thus, Forward Engineering was employed to allow the Spring Testing/Student Orientation system to meet the added requirement.

Since the requirement was for a completely new test, a source of information was required. Fortunately, the professor that was the end-recipient of the grades for the new English test (as well as the existing) was able to provide metrics for the English exams. The English tests offered were in the form of essays, which were graded not by the scanner, but by various English professors. Once the essays were graded, the professors bubbled in forms to indicate student number and score on the essay. The Spring Testing/Student Orientation was then to indicate which class or classes (if any) the student was to receive credit for and then produce a report for the English Department. As there were now two English tests (the old and the new), it was decided that the newly-engineered portion should handle both of them, rather than having the two tests handled by separate pieces of software.

The next step in renovating the Spring Testing/Student Orientation software was to choose a language to program the new sections of code in. In accordance with the previously stated goals for renovation, it was desirable that the renovated system be as simple as possible.

As the Spring Testing/Student Orientation system performed primarily file I/O operations, utilizing a language that handled reading and writing in a "friendly" manner had a high-degree of desirability. So, while C++ had been employed for the initial software, it did not seem to be a good candidate for the renovation language. Python[12] was selected as the language for renovation as it has a reputation for being easy to learn. Additionally, Python handled the necessary file operations in a suitably simple fashion.

The format of the input was pre-decided by the forms fed into the scanner and the files the scanner produced. The scanner produced a text file containing lines with two items each; a student number and a score. In addition, a RSVP file containing information on all the students taking tests was provided

[12] http://www.python.org

before each Spring Testing and Student Orientation, allowing for the correlation of student numbers and student information.

Figure 7.1: Example of selected fields from an unmodified RSVP file

Mass Assign Field	ID	Event ID	Event Mtg	Attendee	Mtg Dt	Status	Evnt Meeting	Last Name	First Name
2385502 000119648 1	[redacted]	000119648	1	00142	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2365255 000119648 1	[redacted]	000119648	1	00003	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2376405 000119648 1	[redacted]	000119648	1	00145	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2394958 000119648 1	[redacted]	000119648	1	00077	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2389305 000119648 1	[redacted]	000119648	1	00045	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2395090 000119648 1	[redacted]	000119648	1	00055	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]
2382587 000119648 1	[redacted]	000119648	1	00021	2010-04-09	Y	Spring Testing 2010	[redacted]	[redacted]

The **ID** field shown in **Figure 7.1** is the single most important field in an RSVP file. Without it, no unique identification of a student is possible. All other fields, including the student name fields, can be missing or incorrect so long as the **ID** field is both complete and correct.

The actual implementation of the new portion of the Spring Testing/Student Orientation software was then initiated via an expanding prototype; a piece of functionality was built and tested, then the next portion was built on top.

The first necessary piece of functionality was to read from the provided RSVP file and glean the ID's of every student that had registered to take tests. Since the Student Numbers were hand-bubbled by instructors, a certain amount of error was expected in the input. Thus, having some manner of checking was highly desirable in accordance with renovation goal **3**. Stripping the RSVP file down to the Student Numbers was easily handled via Python as seen in **Figure 7.2.**

Figure 7.2: RSVP header strip operation

```
id_list = []     #id_list is going to contain every test taker's ID gleaned…
rsvp.pop(0) #Get rid of the headers from the RSVP file.
for line in rsvp:
    id_list.append(int(line.split(',')[0])) #Split the lines by commas…
```

The next important functionality was to check for invalid or malformed student numbers, to attempt to catch human error. Once again, the method for performing the desired functionality was exceptionally simple.

Figure 7.3: Checking for invalid ID's

```
if id not in id_list:   #If the ID from the current working test isn't in…
print 'Invalid ID detected. Check ' + str(id) + '\n'
```

It should be noted that the *not in* operator utilized in **Figure 7.3** is not particularly efficient; a linear search will be performed for each student that took the test. Fortunately, the total number of students is always small (never exceeding the low-hundreds) and as such performance did not have a high priority in the development of the renovated software. Thus, the inefficiency incurred was deemed an acceptable tradeoff for simplicity.

The metric for grading the tests was handled via a pair of arrays, with the student's score (an integer between zero and five) acting as the key.

Figure 7.4: Sample grading metric array

```
key = ['ENGL 100--no credit awarded, must enroll in ENGL 100',
       'ENGL 150--no credit awarded, must enroll in ENGL 150',
       'ENGL 1156--no credit awarded, must enroll in ENGL 1156',
       'ENGL 1157--no credit awarded, must enroll in ENGL 1157',
       'ENGL 1158--3 hours of credit for 1157 awarded, must take ENGL 1158',
       'Exempt--6 hours credit for 1157 and 1158 awarded']
```

For example, if a student had a score of three, then **key[3]** in **Figure 7.4** would indicate their placement and the student would be placed in English 1157.

This encompassed all of the necessary functionality to meet the new requirement, and as such the resulting output was written to a text-file for later use.

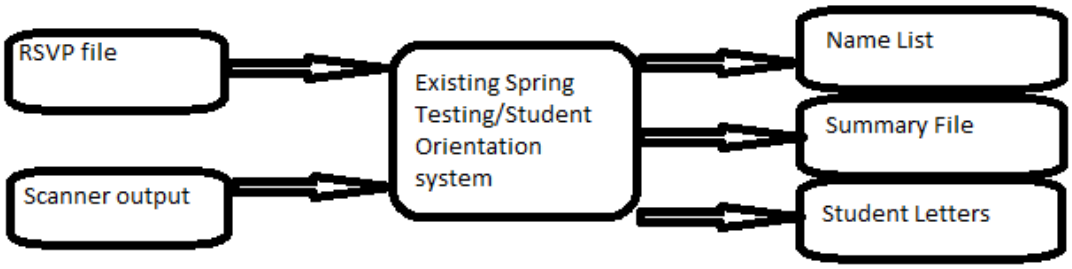
7.2 Mixed Engineering

Reverse Engineering is "the process of discovering the technological principles of a device, object or system through analysis of its structure, function and operation."[13] Reverse Engineering is useful for renovation cases in which there are insufficient sources of information about the target system available. Reverse Engineering can be used as a gap-filler in the knowledge base in such cases.

During a renovation it may be necessary to combine elements that have been forward engineered with those that have been reverse engineered. For the purposes of this document, such a combination will be referred to as Mixed Engineering.

The Spring Testing/Student Orientation system employed by Testing Services suffered from tight-coupling; alterations to one portion of the source code would cause failures in other portions of the system. The particular problem was the coupling of the test scoring system to the report generation system. With the replacement of the English grading portion, the report-generating portion of the Spring Testing/Student Orientation software ceased to function, as it had no provision for the addition of tests.

Figure 7.5: Workflow of pre-renovation Spring Testing/Student Orientation system



[13] http://en.wikipedia.org/wiki/Reverse_engineering

As seen in **Figure 7.5**, there was no separation between test-scoring and report-generating sections of the pre-renovation system, despite there being a logical separation in their operations. Thus, when the test-scoring portion failed to meet requirements, the report-generating portion did as well.

To ensure that the system again met all requirements, reverse engineering had to be employed. The first step in renovating the report generation portion of the system was to understand what manner of input it accepted and what the reports it then generated looked like. The documentation for the Spring Testing/Student Orientation system contained examples of some of the output, and thereby provided a template for the final reports. The documentation did not, however, describe what sort of output was created by the test-grading portion of the software, and thus did not describe what manner of input the report generating portion of the system accepted.

Further investigation revealed that some of the files created from runs of the Spring Testing/Student Orientation system still existed. This provided this necessary information needed to piece together the interactions between the grading and reporting sections of the system. The next step in the renovation was to assess the viability of altering, rather than replacing, the report-generation code. While it would have been preferable to preserve the existing code, several factors made code conservation unfeasible:

1. The system was extremely tightly coupled.
2. The source code was almost entirely uncommented.
3. The existing code had a higher degree of complexity than was desirable.

With these factors considered, it was determined that replacing the report-generation portion of the system was the correct renovation method to employ. As the report-generation portion was doing almost exclusively file I/O operations, Python was again employed for the renovation work.

Rather than attempting the entire replacement of the report-generation software, a series of expanding prototypes was utilized. The first functionality to be reverse engineered in the prototype was a data-gathering section. This section read from an existing, provided, RSVP file (**Figure 7.1**) to compile information on each student for use in the final reports. The RSVP file was saved as CSV (comma-delimited Excel format) and was supposed to be formatted with the columns displayed in **Figure 7.6**.

Figure 7.6: RSVP file column header ordering

```
ID, Status, EmplID, NID, Attendee #, Last Name, First
Name, Address, City, State, Zip, Admit Type, Admit Term, Acad Prog
Status, Acad Prog, Acad Plan, Sub-Plan, ACT COMP, ACT ENGL, ACT MATH,
SAT COMP, SAT VERB, SAT MATH, Telephone, Email ID
```

Figure 7.6 does not, however, accurately depict how the RSVP file was actually formatted on receipt, and the actual column order and total seemed to have been different in each previous instance of Spring Testing and Student Orientation. As there was no way to enforce the standard format for the RSVP file depicted in **Figure 7.6**, or guarantee that all information would be present in the RSVP file, the renovation goal **3** (robustness) was a particular concern. Renovation goal **1**(simplicity) further dictated that rather than attempt to discern precisely what data was available and then alter the reports accordingly, the system should simply allow accept blank fields within the Excel file. The Student Number field, however, was an exception to this robustness.

Figure 7.7: Student number extraction operation

```
for line in rsvp:
        working_student = []    #This is going to be a list of …
        working_student.append(line)    #Append the RSVP info.
        working = line.rstrip('\n').split(',')  #Strip off the …
        working_id = int(working[0])    #Grab the student number (ID).
```

As can be seen in **Figure 7.7**, the RSVP data is associated with a student and the Student Number is extracted from each line in the RSVP file. The Student Number field acts as a primary key, uniquely

identifying each student attending Spring Testing or Student Orientation. As long as this field is non-empty, the system will produce a report for the indicated student. It should also be noted that the algorithm does not check to see if the student number is non-empty. Since no data standard can be enforced, the each RSVP file must be hand-checked and modified before it can be utilized. It is during this modification process that missing ID's are noted, requested, and added.

Once the ability to gather student data from the RSVP file was programmed and tested, it was time to expand the prototype to gather data from the various test result documents. The test result documents fall into two categories; English and non-English exams. The English results were the result purely of renovated code, and thus exactly what manner the data contained therein was structured and contained was known and documented. Since both English tests used the same format, a single algorithm was sufficient to strip data from them. A second algorithm was utilized to collect the data from the non-English (Computer Science, Math, Spanish, etc.) exams.

Since the end reports are by student, and not by exam, the renovated prototype was designed to compile all the information about a single student, then move on to the next.

Figure 7.8: English exam algorithm sample

```
for score_line in file_data:
    score_line_breakdown = score_line.rstrip('\n').split('\t') #Strip off…
    score_line_id = int(score_line_breakdown[2])     #Grab the student…
    if score_line_id == working_id: #See if the current line of the…
        line_accumulator = test_name + ' '  #Going to compile…
        count = 3
        line_len = len(score_line_breakdown)
        while count < line_len:
            line_accumulator = line_accumulator + score_line_breakdown[count]
             + ' '
             count = count + 1
        working_student.append(line_accumulator)     #Append the compiled…
```

The algorithm for the English exams, **Figure 7.8**, starts by breaking down each line in the file by using the 'tab' character as a delineation and then grabs the student number to see if the particular result

matches up to the student about whom information is currently being compiled. Should the student numbers match, the information about the particular test is appended to the current student's data.

Figure 7.9: non-English exam algorithm sample

```
count = 0
while count < 4:
file_data.pop(0)
count = count + 1
for score_line in file_data:    #Finished header strip operation.
    score_line_id = int(score_line[20:29])  #Substring operation to get…
    if score_line_id == working_id: #working_id is from the current line…
        line_accumulator = test_name + ' ' + score_line[31:]    #Substring…
        working_student.append(line_accumulator) #Append the compiled…
```

The algorithm for the non-English exams, **Figure 7.9**, starts by stripping off header information in the various test files. It then proceeds to mine out the student number for comparison. Unlike the English algorithm, however, the non-English uses a substring operation to accomplish this task. This is less robust than the split command used in the English algorithm, but using a split command would have added complexity to the code due to the nature of the exam data produced by the existing system. As the non-English tests had not had any changes in the nature of the questions or the metrics used to grade them in at least several years, it was deemed that there was a low risk of the substring operations failing due to exam changes and that the tradeoff between simplicity and robustness was acceptable.

Both algorithms have a high-degree of simplicity, which is in line with stated renovation goals. Neither algorithm is particularly efficient, however, as the algorithms perform linear searches and mining operations on every line of each of the test result files for each student that signed up for exams. Fortunately, the system does not handle more than several hundred students, meaning that the entirety of the operations is executed in a reasonable amount of time.

7.3 Reverse Engineering

Once the Spring Testing/Student Orientation renovation prototype was expanded to include the necessary data gathering and linking operations, it became necessary to add on the algorithms to generate the various final reports. Unlike the previous section, which was a mix of forward engineering new data handling algorithms and reverse engineering algorithms to handle the data produced by the un-renovated portion of the system, the report section was almost entirely a reverse engineering problem.

Rather than develop new styles of report to meet the output requirements of the Spring Testing/Student Orientation, the challenge was to recreate the reports generated by the existing software. The first report to be recreated by the prototype was a name list. The name list was simply a text file containing the names and results of each student who had registered for Spring Testing or Student Orientation. The format was simple, and sufficient examples existed from previous runs to allow for the reverse engineering of an algorithm to produce the desired format.

Figure 7.10: Name list generation algorithm

```
name_list = open(path + 'name_list.txt', 'w')    #START creating name list.
    for line in students:   #For every student, write their name and their…
        rsvp_line = line[0].split(',')
        name = rsvp_line[3] + ', ' + rsvp_line[4]   #Student name, last and…
        name_list.write(name + '\n')    #Write the name.
        count = 1
        line_length = len(line)
        while count < line_length:  #Skip the RSVP line, but write all the…
            name_list.write(line[count] + '\n')
            count = count + 1
        name_list.write('\n\n\n')   #Skip some lines between students.
    name_list.close()   #END creating name list.
```

As with previous examples, and in line with the stated renovation goals, the name list generation algorithm shown in **Figure 7.10** is very basic in its structure. It was also required that the name list should contain not just all of the students, but also have them ordered alphabetically. As the provided

RSVP file already contained the students sorted in the desired manner, and the student data was mined directly from the RSVP file, the desired order already existed within the data and a linear operation was sufficient to produce the correct ordering in the final report.

Of particular concern were the array access operations utilized in this algorithm. Hard-coding precisely where to find specific information within the RSVP file ran contrary to the stated desire for robustness, as a change to the format of the RSVP file would produce unanticipated results. However, due to the unstable nature of the RSVP file formats, each RSVP file was already being hand-modified to ensure standardization prior to use, thus mitigating the risk that a malformed RSVP file represented.

The second of the three reports that needed to be generated was a series of summary files. Unlike the other two reports, there were no files of this type left over from previous Spring Testing or Student Orientation runs. Additionally, the documentation contained no example of the format the summary files should conform to. The only information available on what the summary files should look like was a brief description that there should be a file for each student, that the files be tab-delimited, and that the file should contain the student's name and test results. To further complicate the matter, no one involved in Spring Testing/Student Orientation seemed to be able to provide information in advance as to exactly how the summary files should be formatted.

An interim algorithm was created that simply wrote a file for each student containing their name and test results, with tab characters separating each unit of information. Post-deployment of the renovated Spring Testing/Student Orientation system, the office of Admissions made some general requests as to the format of the summary files. The summary file generating algorithm was further refined to bring the reports in line with what Admissions requested, and the final version appeared as shown in **Figure 7.11**.

Figure 7.11: Summary file generation algorithm

```
summary_file = open(path + 'Summary_File.txt', 'w') #START creating summary…
    for line in students:   #For each student, create a tab-delimited file…
        rsvp_line = line[0].split(',').rstrip('\n')  #Break up the RSVP line…
        student_number = rsvp_line[0]
        summary_file.write(student_number + '\t')  #Write the student number…
        count = 1
        line_length = len(line)
        while count < line_length:  #Once again, skip the RSVP line, but add…
            working_item = line[count]
            if (working_item == 'ENGL_ADV') or (working_item == 'ENGL'):
                working_item = 'ENGLISH'
            summary_file.write(line[count] + '\t\t')
            count = count + 1
        summary_file.write('\n')
    summary_file.close()    #END creating summary file.
```

The algorithm shown in **Figure 7.11** is incomplete, only producing a partial summary file. The request made by Admissions included that, for each student, a listing of what courses that particular student received credit for should appear. However, one of the English exams grants only contingent credit. For example, a student might receive credit for English 1157 so long as they take and complete English 1158 with a grade of C or higher on the first attempt. Should the student fail to receive the necessary grade, the contingent credit for English 1157 would then be withdrawn, forcing the student to take 1157. The contact within the Office of Admissions was queried as to precisely how to handle the contingent credit, but a response was never received on the matter. With no information forthcoming, renovation on this particular section of the Spring Testing/Student Orientation system was abandoned.

The final report type of the three was a letter format. One file for each student was to be created. These files were printed out on the day of the exams and handed out to the students. The letter-creation algorithm needed to mimic the format of the letters produced by the pre-renovation system as closely as possible, though absolute precision in the mimicry was unnecessary.

Figure 7.12: Student letter generation algorithm

```
for line in students:   #For each student, produce a formatted letter.
        rsvp_line = line.pop(0).split(',')  #Split the RSVP file by commas…
        working_file = open(path + '/letters/' + rsvp_line[0] + '.txt', 'w')
        working_file.write('University of New Orleans\nStudent Orientation '
            + year + '\n\n\n\n\n')
        working_file.write('\t' + rsvp_line[3] + ', ' + rsvp_line[4] + '\n')
        working_file.write('\t' + rsvp_line[5] + '\n')
        working_file.write('\t' + rsvp_line[6] + ', ' + rsvp_line[7] + ' ' +
rsvp_line[8] + '\n\n\n\n')
        working_file.write('\t\t\tUniversity of New Orleans\n')
        working_file.write('\t\t\tStudent Orientation ' + year + '\n')
        working_file.write('\t\t\tTEST REPORT FOR:\n')
        working_file.write('\t\t\t' + rsvp_line[3] + ', ' + rsvp_line[4] +
'\n')
        working_file.write('\t\t\tUNO Student ID: ' + rsvp_line[0] + '\n')
        working_file.write('\t\t\tLevel: ' + rsvp_line[10] + '\n')
        working_file.write('\t\t\tProgram: ' + rsvp_line[11] + '\n')
        working_file.write('\t\t\tPlan: ' + rsvp_line[12] + '\n\n')
        working_file.write('\t\tACT--> E: ' + rsvp_line[26] + '\tM: ' +
rsvp_line[25] + '\tComp: ' + rsvp_line[24] + '\n')
        working_file.write('\t\tSAT--> V: ' + rsvp_line[29].rstrip('\n') +
'\tM: ' + rsvp_line[28] + '\tComp: ' + rsvp_line[27] +      '\n\n\n')
        for sub_line in line:
            working_file.write(sub_line + '\n')
        working_file.close()
```

In order to allow for easy testing and modification to the student letter generation algorithm shown in **Figure 7.12**, each line in the final letters was left as a single write operation, rather than combining the entire formatting operation into a single write. For each student, a file is produced containing a standard header, information from the RSVP file (which may be blank), and results from all, if any, exams the student took during Spring Testing or Student Orientation. As with the name list, the use of hard-coded locations for accessing specific units of information from the RSVP file posed a risk to robustness, but was deemed to be outweighed by the need for simplicity.

7.4 Results of Renovation of the Spring Testing/Student Orientation System

The end result of the Spring Testing/Student Orientation system renovation was software containing a preserved portion of the original C++ system coupled with Python portions handling the grading of the English exams and the production of the final file output. It should also be noted that because only an English exam is offered for non-Spring Testing Student Orientations, in most cases only the renovated portions of the system are utilized.

Figure 7.13: Renovated Spring Testing structure

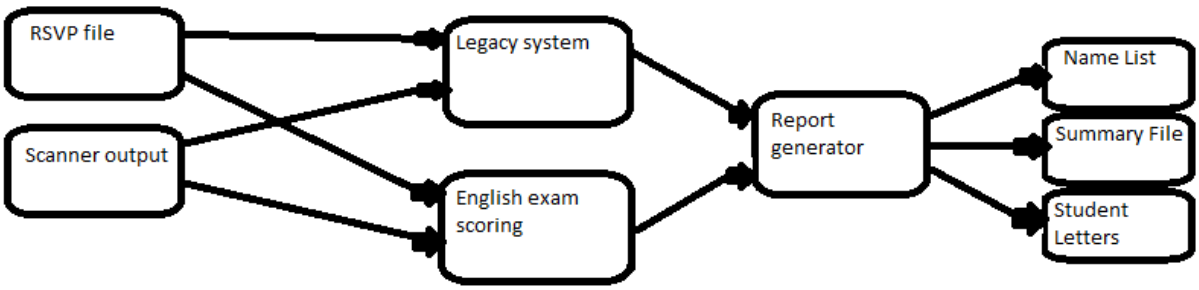
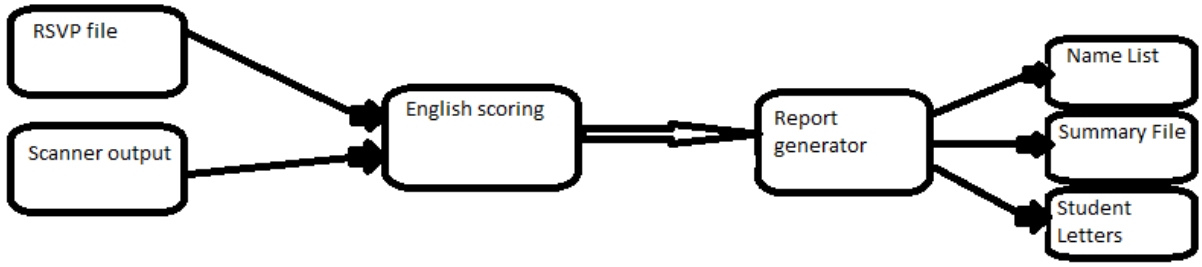


Figure 7.14: Renovated Student Orientation structure



As can be seen in **Figures 7.13** and **7.14**, the overall structural complexity of the Spring Testing superset was increased during renovation. This was in direct opposition to renovation goal **1** (simplicity). However, the renovated system did, in this case, reuse a portion of the existing Spring Testing/Student Orientation system, which coincided with renovation goal **4** (preservation).

In the case of the Student Orientation subset, the structural complexity did not show the same kind of increase. However, no portion of the original system was preserved in this case. In both cases, the resulting system had a lower degree of coupling, putting the renovation in line with stated goal **3** (robustness).

In both cases, the renovated system was capable of fulfilling the grading requirements, fulfilled two out of three of the report requirements, was reasonably user-friendly, and completed all tasks in a reasonable amount of time. As the Office of Admissions had ceased to communicate about the now-abandoned report format, the only failed requirement for the Spring Testing/Student Orientation system is mitigated. Thus, on the whole, the renovated system can be seen as a moderate success.

8. Integration

To Integrate is to “to form, coordinate, or blend into a functioning or unified whole (Integrating, n.d.).” Software Integration is then the coordination or blending of components, process, and workflows into a working whole. Integration becomes a key factor in Software Renovation when adjustments are made to existing workflow and/or partial replacement of existing systems occurs.

8.1 Integration Case Study: Literary Rally

The University of New Orleans’ Office of Admissions annually hosts Literary Rally, which is an event wherein local high school students register for various tests. Each test is treated as a separate event, and several winners are chosen depending on district and total number of students taking the test.

The Office of Testing Services had been responsible for grading the tests and compiling results for the Office of Admissions. The general requirements for the Literary Rally system were:

1. To read from a provided list and associate students with tests, to ensure that only students that had signed up for a specific test were allowed to take that test.
2. To grade tests based on provided keys and sort final results by score.
3. To handle tie-breaking and multiple winners based on districts.
4. To operate and complete all tasks in a short amount of time, as results were desired on the same afternoon in which the tests were given.
5. To be robust and well-documented to prevent need for any debugging on test-day.

However, the existing Literary Rally software employed by Testing Services was no longer meeting these requirements. Notably, the existing Literary Rally software was undocumented in both structure and operation. Further, the software was not easy to use and did not display errors involving mistakes within the input file when run outside of Visual Studio. In addition, the methodology behind the actual scoring

and evaluating process was arcane and had apparently changed from year to year. For these reasons, it was deemed necessary to replace the existing Literary Rally system. To reach this renovation destination, however, there were two important factors: communication and workflow.

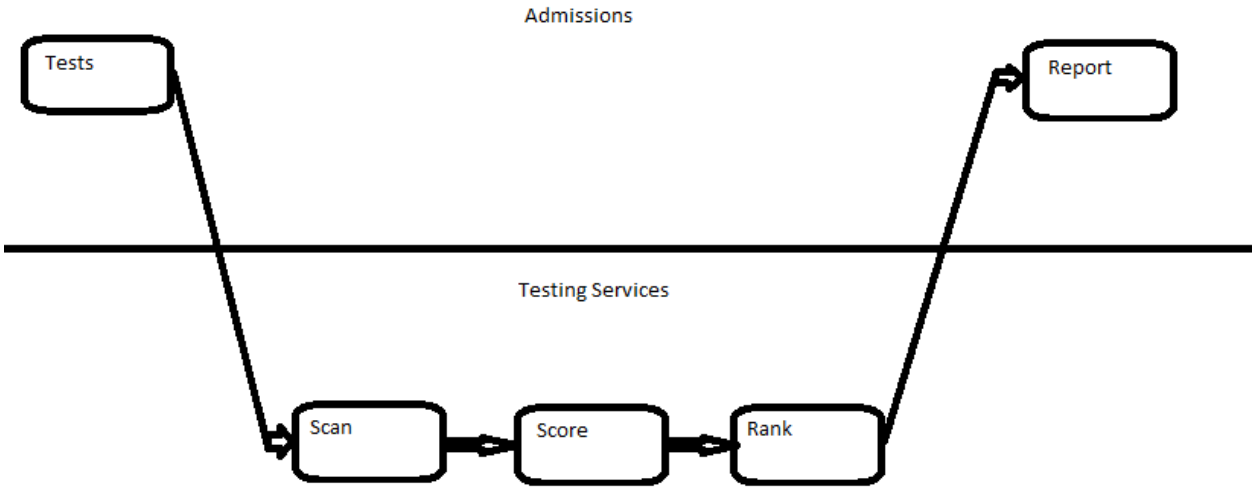
8.1.1 Communication

Communication initially proved to be a challenge when renovating the Literary Rally system. Before any work could be done on the system a set of requirements had to be ascertained from the Office of Admissions. However, no formal requirements had ever been documented. Thus, new requirements had to be gleaned over the course of several meetings with personnel from Admissions. The tie-breaker requirement quickly proved to be a sticking point in the process; specifically, the Office of Admissions had at no point in the past developed a concrete and coherent policy on how the tie-breakers should be handled, and had apparently resorted to tie-breaking by hand in previous years. Also, there was no solid information on how exactly winners were broken down by district. With these problems in mind, a dialogue was started with a representative for Admissions' IT capabilities and a change to the workflow was agreed upon.

8.1.2 Workflow

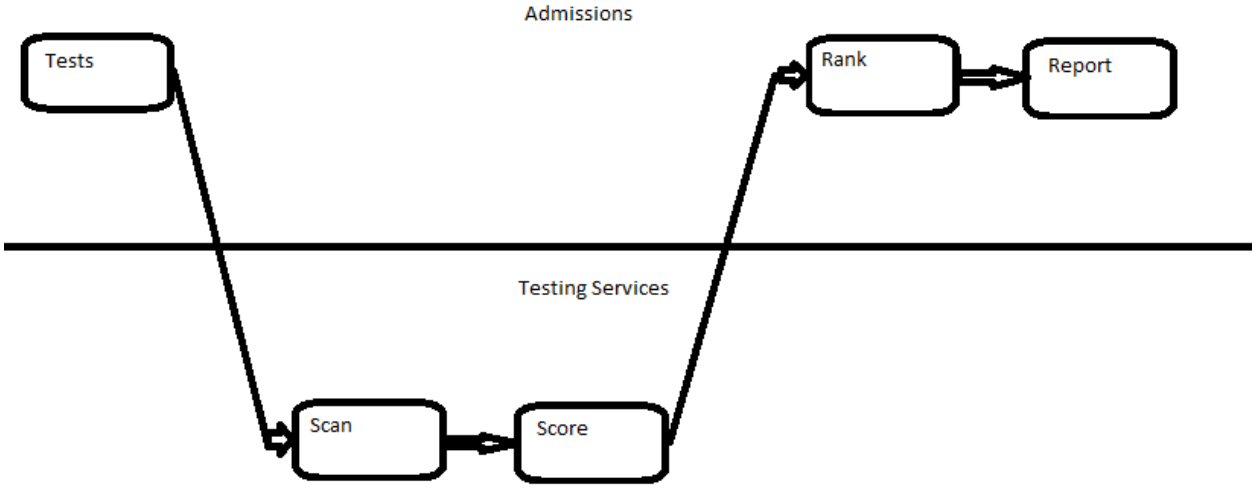
Because Admissions was in a position to decide how tie-breakers would be handled and had knowledge of how the district winners were picked, it was negotiated that Testing Services would score the tests before providing Admissions with the compiled results, and that Admissions personnel would then make the final decisions on winners. Thus, the workflow changed from:

Figure 8.1: Original Literary Rally workflow



To:

Figure 8.2: Renovated Literary Rally workflow



To achieve the alteration in workflow shown between **Figure 8.1** and **Figure 8.2**, a format for the data was negotiated so that the Admissions personnel would be able to easily dump results files into Excel

files that were constructed by Admissions for deciding winners in the non-tie cases. Further, a directory for the output files was negotiated so that Admissions would have easy access to the results.

Due to the change in workflow, a portion of the existing Literary Rally system could be immediately discarded. A portion, however, would still need to be renovated to restore necessary functionality to the Literary Rally process. At the time of the renovation, the existing software was failing all requirements, both existing and new. As part of the existing system was no longer necessary, and the entirety of the Literary Rally software had failed to function, it was decided that it would be replaced.

As a line of communication had been established with the Office of Admissions, a basic methodology for renovating the Literary Rally process could be determined. Firstly, the format of the incoming tests was documented. Each test had a key sheet that was to be provided in advance by Admissions. The answer sheets themselves were to contain an identification number, and the student's answers. After some negotiation, it was decided that asking students to not leave questions blank was a reasonable request, as the tests were all multiple-choice in format and the penalty for a wrong answer was no worse than the penalty for a blank answer.

The next piece of information gleaned from the communication with the Office of Admissions was that they would be able to provide listings of the students that had signed up for exams as well as which exams were being offered. These listings were particularly important, as students were only supposed to be allowed to take exams that they had signed up for, and there was a need to match test results with names and ID's.

The work the renovated Literary Rally system was to perform was split into two pieces; a preprocessor that would be run before Literary Rally, and a system to actually grade the tests on the day of the exams. The listing file the preprocessor was to work with was a comma-delimited Excel format, and the

first step was to utilize this file to produce a list of all the exams. Since the renovated system was to perform file I/O operations, Python was utilized for the Literary Rally renovations as well.

Figure 8.3: Exam list compilation algorithm

```
test_list = []  #This will contain the test names.
for line in raw_list:   #Time to dig out the test names.
        working_line = line.split(',')  #Students.csv is comma-delimited, so…
        test = working_line[5]  #Get the test name from this row.
        if test not in test_list:   #Check to see if the test is already in…
            test_list.append(test)  #If it isn't in the list, add it.
```

The algorithm shown in **Figure 8.3** performs a linear search and compiles a list of all the tests found in the provided file. While this is not an efficient algorithm, the preprocessor is meant to be run as early as several days before Literary Rally, and the file it operates on is only a few hundred lines, so efficiency had almost no priority in the design. More importantly, the algorithm is extremely simple, putting it in line with renovation goal **1**.

The next step in the preprocessor had an even higher degree of simplicity.

Figure 8.4: Work order production

```
for line in test_list:  #Time to write the work order listing.
        work_order_out.write('LR' + str(work_count) + '     ' + line + '\n')
        work_count = work_count + 1
```

The short loop shown in **Figure 8.4** produced a file containing pairs of work orders and test names. The work order file was then utilized on test day so that a naming convention would be followed and test results would not be ambiguously named or confused for the wrong exam.

The final step in the preprocessor was to produce a listing of exams with the ID's of the students that were signed up to take the tests.

Figure 8.5: ID list compilation

```
for test in test_list:  #Now to put together the tests and test ID's.
      working_file = open(path + 'LR' + str(work_count) + '.txt', 'w')
      work_count = work_count + 1
      for line in raw_list:   #Find everyone who is taking the currently…
            working_line = line.split(',')  #Students.csv is comma-delimited…
            if working_line[5] == test: #If the selected line matches a…
                id_clean = str.replace(working_line[2], ' ', '')    #Remove…
                test_clean = str.replace(test.rstrip('\n'), ' ', '_')
                working_file.write(test_clean + '\t' +
working_line[0].lstrip('"').strip() + ',' +
                      working_line[1].rstrip('"').strip() + '\t' + id_clean +
                      '\t' + working_line[3] + '\t' + working_line[4] + '\n')
```

Once again, the algorithm shown in **Figure 8.5** traded efficiency for simplicity by performing a linear search of the listing file for each of the exams being offered. The algorithm operated by splitting the working line using commas as the delimiter, then checked to see if the test ID matched the test for which student ID's were being compiled. If there was a match, cleanup would be performed to remove quotation marks and other unnecessary characters, then the information would be written to file for use on the day of Literary Rally.

On test day, the various test forms were brought in as they were completed. When a stack of answer sheets arrived, the key sheet would be placed on top and the entire stack would be run through the optical scanner. Each stack was named according to the work order numbers produced by the preprocessor. Ideally the files outputted by the scanner would then be of the form shown in **Figure 8.6**.

Figure 8.6: Scanner output format

```
####################...
xxxxxxx         ####################...
xxxxxxx         ####################...
```

In **Figure 8.6**, *xxxxxxx* would represent an ID number, while ******** would represent a linear series of numbers corresponding to the answers. Functionally, both would be a string composed of integer characters. The first line would have no ID because it is the key, and thus would not correspond to a student.

The first major step in the test day software was to check to see if the data produced by the scanner was properly formed. To accomplish this, the entire file was read in, the key was popped off and saved, and each line was split using the space character as the delimiter. From that point, some error checking could be performed.

Figure 8.7: Error checking

```
if len(working_line) > 2:
            print 'WARNING!!! Omitted question or malformed ID detected!
Check paper for student ' + working_id + '.\n'
            error_log.write('WARNING!!! Omitted question or malformed ID
detected! Check paper for student ' + working_id + '.\n')
            failure = 1
            break
if len(working_line) == 1:
            print 'WARNING!!! Malformed data detected! Check sheets for
missing ID or answers.\n'
            error_log.write('WARNING!!! Malformed data detected! Check sheets
for missing ID or answers.\n')
            failure = 1
            break
 if working_id not in id_list:  #Failure case, code terminates if this…
            print 'WARNING!!! Invalid ID detected! Check ID ' + working_id +
'.\n'
            error_log.write('WARNING!!! Invalid ID detected! Check ID ' +
working_id + '.\n')
            failure = 1
            break
 elif working_id in used_ids:   #Time to see if the ID is a duplicate, also…
            print 'WARNING!!! Duplicate IDs detected! Check IDs ' +
working_id + '.\n'
            error_log.write('WARNING!!! Duplicate IDs detected! Check IDs ' +
working_id + '.\n')
            failure = 1
            break
 else:
            used_ids.append(working_id) #This is (probably) a valid ID, so…
```

The algorithm shown in **Figure 8.7** was utilized to break down the five detectable cases the scanner output could generate. In the first case, if the length of the array produced by splitting the working line was greater than two, then there was the potential that the student had left a gap or filled in data that should not be there. In the second case, the length of the array produced by splitting the working line was one, which would then tend to indicate that data was missing. This was usually the result of a student not bubbling in his or her ID. In the third case, the ID of a student taking the exam did not match any of the students on the list to take that exam, possibly indicating cheating but more likely indicating an incorrectly bubbled ID. In the fourth case, a duplicate ID was detected, usually indicating an incorrectly bubbled ID. In the fifth case, a valid, non-duplicate ID was detected. The ID would be added to the list of used ID's, to detect duplicate ID's in later lines. It is worth noting that the fifth case would not necessarily indicate a valid ID. For example, if a student had failed to take an exam he or she had signed up for, and another student incorrectly bubbled in his or her ID, and the malformed ID happened to match up to the missing student's ID, there would be no way for the system to detect the error. Fortunately, the chances for such an error are relatively minor, and it was deemed and acceptable risk.

Scoring the tests was, in theory, exceptionally simple. Each answer string would be compared character-by-character to the key string. As students were not supposed to leave blank answers, and guessing if the answer were not known would produce at least some chance of the correct answer being marked, it was assumed that the second item in the array produced by splitting the line on the space character would be the entire answer string. During Literary Rally, however, it became clear that students quite frequently left blank answers, even though they had been instructed not to and it was in their best interests not to do so.

Initially, the number of blank answers seemed disastrous. Every file had to be hand-modified to correct for blank answers. Several factors mitigated the problem this presented, however. First, all the files had

to be hand-modified regardless of blank answers. Inevitably students would incorrectly bubble in ID's. The incorrect ID numbers took longer to track down and correct than the blank answers, which were easily spotted and fixed. Even with the hand-modifications to the scanner-produced files, the Testing Services crew was able to score the exams faster than the Admissions crew could tie-break and sort out the winners. For these reasons, the design was retained, despite the extra work required, as it was exceptionally simple in structure, required very little operator interaction to run, and was robust enough to handle all of the exams, regardless of size variations.

There was, however, a need to properly detect answer gaps in the scanner-produced files. If questions were left blank in anything other than the first or last positions, the first error case when checking for malformed ID's would detect the omissions. Since most of the cases were taken care of by the initial design, a further piece of code was added to check for missing answers in the first or last places, as shown in **Figure 8.8**.

Figure 8.8: Additional error checking

```
if len(student_answers) < number_of_questions:
      print 'WARNING!!! First or last question omitted! Check ID ' +
working_id + '.\n'
      error_log.write('WARNING!!! First or last question omitted! Check ID '
+ working_id + '.\n')
      failure = 1
      break
```

It should also be noted that in every error case, a variable named *failure* is given the value of 1, and a *break* command is called to cause the Literary Rally system to skip the remaining scoring process. These operations are performed to prevent unnecessary processing and to ensure that no file is written containing partial results that then might be confused for a successful run. The Literary Rally system either produces complete results, or no results at all and error messages to help locate the detected error in the scanner-produced file.

8.1.3 Results of Literary Rally integration case study

The end result of the renovation of Testing Services' Literary Rally system was the complete replacement of the existing software system and an overhaul of the day of Literary Rally workflow. Prior to renovation, scanned files would be created, processed, scored, and ranked by Testing Services. Post-renovation, the ranking operations were handled by the Office of Admissions, though the pre-ranking workflow operations remained unchanged.

Prior to the renovation, the Literary Rally system had failed to produce workable results during the previous two years. As such, all scoring and ranking had to be done by hand and the results were not available in the desired time frame. After the renovation, far less work had to be done by hand, and results were produced on time. Further, the renovated system left records intact that could be referenced should there have been a problem detected in the ranking process at some later date. No such records were left when Literary Rally was done by hand.

The renovation of the Literary Rally system was also in line with the stated renovation goals. There were only a combined total of 115 lines of code in the two pieces of Literary Rally software. Additionally, the structure of the source code was extremely basic, the code was well-documented, the code was robust enough to allow for variations in tests from year-to-year, and common errors were explicitly communicated to the user. These qualities reflect renovation goals **1** (simplicity), **2** (documentation), and **3** (robustness).

The renovated system was, however, less in-line with renovation goal **4** (preservation). No part of the original Literary Rally system was conserved, and its interface was not replicated. One of the end report types was also not supported and was abandoned in the final version of the renovated software.

In summary, the renovated Literary Rally system was not ideal, and did sport some flaws primarily as the result of communications breakdown, but did allow the Literary Rally process to meet necessary requirements and thus was considered an overall success.

9. Conclusions

The Office of Testing Services had particular needs that shaped the renovation process. While each case has its own unique functional requirements, they all share the same functional requirement: simplicity and robustness trumped the need for efficiency as there was no guarantee that qualified staff would be on hand at any given time, and expected staff turnover was very high due to the employment of student workers. An extreme form of the Keep It Simple Stupid (KISS) methodology was employed to this end. All replacement code was created with two principles in mind:

1. Minimize the need for training.
2. Document thoroughly.

Each of these principles helped to enforce ease of code understanding. Principle 1 was aimed at cutting down the effort a new worker would have to in understanding the code. As there was no guarantee what sort of background a new worker might have, it was desirable that gaining an understanding of the renovated code require as little computer science knowledge and experience as possible. Thus, in accordance with this principal, no advanced structures were utilized; no optimized searches, no hash tables, and no functions. All renovated code was written to be linear, easy to follow, and utilizing only basic structures. The final step to ensure ease of code understanding was to comment all algorithms thoroughly, as well as to document the function and proper operation of each system. Python performed very well for these purposes, producing simple, easy-to-read code.

On the whole, the renovation of the systems employed by the Office of Testing Services was successful. The renovated systems have been better documented, and are easier to understand. For four months, the system was operated and managed smoothly by a graduate assistant with a business background.

References

Bing, S. (2000). Oh, Sure. Now They're Sorry Y2K idiots cost business $500 billion! Is no one to be punished? *Fortune Magazine*. Retrieved from http://money.cnn.com/magazines/fortune/fortune_archive/2000/02/07/272831/index.htm

Lewis, G. A., & Morris, E. J., & Smith, D. B., & Simanta, S. (2008). SMART: Analyzing the Reuse Potential of Legacy Components in a Service-Oriented Architecture Environment. Retrieved from http://www.sei.cmu.edu/reports/08tn008.pdf

Dickens, T. (2002). Migrating Legacy Engineering Applications to Java.

Everaars, C.T.H., & Arbab, F., & Koren B. (2004). Modernizing Existing Software: A Case Study.

Integrating. (n.d.). In *Mirriam-Webster.* Retrieved from http://www.merriam-webster.com/dictionary/integrating?show=0&t=1287984961

Suganuma, T., & Yasue, T., & Onodera, T., & Nakatani, T. (2008). Performance Pitfalls in Large-Scale Java Applications Translated From COBOL.

Vita

The author was born in New Orleans, Louisiana. He obtained a Bachelor's degree in computer science from the University of New Orleans in 2007. He joined the University of New Orleans computer science graduate program to pursue a Masters degree in computer science, and became a graduate student under Dr. Shengru Tu in 2008.